Introduction to SQL

Class 18

Course Overview

- Introduction to SQL
  - Databases, Tables
  - Classification of SQL – DDL, DML, DCL, TCL
    - DDL – CREATE, ALTER, DROP
    - DML – SELECT, INSERT, UPDATE, DELETE
    - DCL – GRANT, REVOKE
    - TCL – COMMIT, ROLLBACK, SAVEPOINT
  - Data types, Operators
  - Keys – Primary, Foreign, Composite, Unique, Alternate
  - Integrity Constraints – Domain Integrity Constrains, Entity Integrity Constraints, Referential Integrity Constraints
  - Joins – Outer Joins, Left Outer Joins, Right Outer Joins, Inner Joins.
  - Queries, Subqueries, Functions, Flow Control (IF, CASE, WHILE, REPEAT, LOOP), ,Stored functions ,Stored Procedures
  - Views
  - Indexes, Cursors, Triggers, Events
  - ACID
  - Concurrency and locking (Implicit locks, explicit locks, row level locks, table level locks, database level locks)
  - Tuning SQL queries and optimizing performance
  - SQL Databases vs NoSQL Databases
  - How SQL databases internally works

Course Overview

- Data modelling and database design
  - What is Data Model – (Steps Involved, Conceptual Data Modelling, Database design)
  - Entity Relationship model
  - Elements of entity relationship model
  - Entities, Subtypes, supertypes
  - Regular, Strong, Weak entities
  - Identifying and modelling entities
  - Relationships – Modelling, defining and types of relationships
  - Minimum and maximum relationships
  - Attributes- Finding attributes, meaningful components for attributes
  - Diagrammatic conventions
  - Transforming entity relationship model into relational schema
  - Logical database design (Top-Down Approach-E/R Modelling, Bottom-Up Approach-Normalization)
  - Difference between top-down and bottom-up approach
  - Normalization
    - Need for normalization
    - Forms of normalization – 1nf,2nf,3nf, Boyce Code Normalization
  - Case studies and discussion
  - Designing few real life databases

Third Normal Form

- A table is in 3NF iff
  - it is in 2NF and
  - all its attributes are determined only by its candidate keys and not by any non-prime attributes

Example

- Table BorrowedBooks

BookNo	Patron	Address	Due
B1	J. Fisher	101 Main Street	3/2/15
B2	L. Perez	202 Market Street	2/28/15

- Candidate key is BookNo
- Patron $\rightarrow$ Address

3NF Solution

- Put address in separate Patron table

BookNo	Patron	Due
B1	J. Fisher	3/2/15
B2	L. Perez	2/28/15

Patron	Address
J. Fisher	101 Main Street
L. Perez	202 Market Street

Another example

- Tournament winners

Tournament	Year	Winner	DOB
Indiana Invitational	1998	Al Fredrickson	21 July 1975
Cleveland Open	1999	Bob Albertson	28 Sept. 1968
Des Moines Masters	1999	Al Fredrickson	21 July 1975

- Candidate key is {Tournament, Year}
- Winner →DOB

Boyce-Codd Normal Form

- Stricter form of 3NF
- A table *T* is in BCNF iff
  - for every one of its non-trivial dependencies $X \rightarrow Y$, *X* is a super key for *T*

EMP_ID	EMP_COUNTRY	EMP_DEPT	DEPT_TYPE	EMP_DEPT_NO
264	India	Designing	D394	283
264	India	Testing	D394	300
364	UK	Stores	D283	232
364	UK	Developing	D283	549

EMP_ID → EMP_COUNTRY

EMP_DEPT → {DEPT_TYPE, EMP_DEPT_NO}

Candidate key: {EMP-ID, EMP-DEPT}

EMP_COUNTRY table:

EMP_ID	EMP_COUNTRY
264	India
264	India

EMP_DEPT table:

EMP_DEPT	DEPT_TYPE	EMP_DEPT_NO
Designing	D394	283
Testing	D394	300
Stores	D283	232
Developing	D283	549

EMP_DEPT_MAPPING table:

EMP_ID	EMP_DEPT
D394	283
D394	300
D283	232
D283	549

```
EMP_ID   →    EMP_COUNTRY
EMP_DEPT   →   {DEPT_TYPE, EMP_DEPT_NO}
```

Candidate keys:

For the first table: EMP_ID
For the second table: EMP_DEPT
For the third table: {EMP_ID, EMP_DEPT}

Advantages of Normalization

- Normalization helps to minimize data redundancy.
- Greater overall database organization.
- Data consistency within the database.
- Much more flexible database design.
- Enforces the concept of relational integrity.

Disadvantages of normalization

- You cannot start building the database before knowing what the user needs.
- The performance degrades when normalizing the relations to higher normal forms, i.e., 4NF, 5NF.
- It is very time-consuming and difficult to normalize relations of a higher degree.
- Careless decomposition may lead to a bad database design, leading to serious problems.

Top-Down Database design vs Bottom-Up Database design

- Top-Down Design: In top-down design, the database design starts with a high-level conceptual model and progresses to lower levels of detail, such as logical and physical data models. This approach is best suited for complex database systems that require a comprehensive understanding of the data requirements before the database is built.
- Bottom-Up Design: In bottom-up design, the database design starts with existing database structures and data and then progresses to higher levels of abstraction, such as logical and conceptual data models. This approach is best suited for organizations that have existing databases and need to integrate or improve them, rather than starting from scratch.

How MySQL Database Internally Works

Overview of MySQL Database Architecture

- MySQL database architecture consists of several components, including the storage engine, query optimizer, parser, and server.
- Each component plays a crucial role in the processing and storage of data in MySQL.

Storage Engine

- The storage engine is responsible for managing data storage and retrieval in MySQL.
- MySQL supports multiple storage engines, including InnoDB, MyISAM, and Memory.
- Each storage engine has its own way of storing and retrieving data.

InnoDB Storage Engine

- InnoDB is the default storage engine for MySQL.
- InnoDB is a transactional storage engine that provides support for ACID (Atomicity, Consistency, Isolation, Durability) properties.
- InnoDB stores data in pages, and each page is 16KB in size.
- InnoDB uses a B-tree index structure to retrieve data efficiently.

MyISAM Storage Engine

- MyISAM is a non-transactional storage engine in MySQL.
- MyISAM stores data in tables that are made up of data files and index files.
- MyISAM is faster than InnoDB for read-intensive operations.

Memory Storage Engine

- Memory is a non-persistent storage engine in MySQL.
- Memory stores data in memory, which means that data is lost when the server shuts down.
- Memory is useful for caching frequently accessed data.

Query Optimizer

- The query optimizer is responsible for determining the most efficient way to execute a query.
- The query optimizer uses statistics about the data to determine the best execution plan.
- The query optimizer can choose different storage engines for different tables in a query.

Parser

- The parser is responsible for analyzing the syntax of SQL statements.
- The parser checks the syntax of SQL statements and generates an internal representation of the query for execution.
- The parser ensures that the SQL statement is well-formed and follows the rules of the MySQL syntax.

Server

- The server is responsible for managing connections, security, and administration of the MySQL database.
- The server handles user authentication and authorization, and it manages the storage engines and query optimizer.
- The server communicates with the storage engine to store and retrieve data.

Optimizing SELECT Queries in MySQL

Optimizing the SELECT Clause

- The SELECT clause specifies the columns to be retrieved.
- Only the necessary columns should be selected to reduce the amount of data retrieved.
- Avoid using the * symbol to select all columns as it can impact performance.

Optimizing the FROM Clause

- The FROM clause specifies the tables to be queried.
- Use indexes on the columns used in the WHERE clause to optimize table access.
- Join tables using the INNER JOIN keyword instead of other join types to reduce the number of rows returned.

Optimizing the WHERE Clause

- The WHERE clause specifies the conditions to filter the data.
- Use indexes on the columns used in the WHERE clause to reduce the number of rows scanned.
- Use the AND and OR operators carefully to avoid creating complex conditions that can slow down the query.

Optimizing the ORDER BY Clause

- The ORDER BY clause sorts the result set in a specific order.
- Use indexes on the columns used in the ORDER BY clause to optimize sorting.
- Limit the number of rows returned using the LIMIT clause to reduce the amount of data sorted.

Using EXPLAIN to Analyze Query Performance

- The EXPLAIN statement can be used to analyze the performance of SELECT queries.
- EXPLAIN displays the query execution plan, including the order in which tables are accessed, the number of rows scanned, and the use of indexes.
- Use EXPLAIN to identify potential bottlenecks in the query and optimize accordingly.

EXPLAIN statement

- id: An identifier for the SELECT statement within the query. If the query involves a subquery, each subquery will be assigned a unique identifier.
- select_type: The type of SELECT statement, such as SIMPLE (a simple SELECT statement), PRIMARY (a subquery in the FROM clause), or UNION (a UNION statement).
- table: The name of the table being accessed.
- partitions: The number of partitions being accessed (if the table is partitioned).
- type: The access method used to retrieve rows from the table. This can include values such as ALL (a full table scan), index (an index scan), or range (an index range scan).
- possible_keys: The indexes that could be used for this query.
- key: The index that MySQL actually used for this query.
- key_len: The length of the index that MySQL used.
- ref: The column or constant used with the index to access rows from the table.
- rows: The estimated number of rows that MySQL will need to examine to execute the query.
- filtered: The percentage of rows that MySQL expects to match the WHERE clause.
- Extra: Additional information about how the query will be executed, such as the use of temporary tables or filesort.

Horizontal Scaling vs Vertical Scaling with Respect to Databases

What is Scaling?

- Scaling is the process of increasing the capacity of a system to handle more load or data.
- Scaling can be achieved in two ways: horizontal scaling and vertical scaling.

Vertical Scaling

- Vertical scaling involves adding more resources to a single server to increase its capacity.
- Resources that can be added include CPU, RAM, and storage.
- Vertical scaling is limited by the capacity of the server and can be expensive.

Horizontal Scaling

- Horizontal scaling involves adding more servers to a system to increase its capacity.
- Resources are distributed across multiple servers, making it easier to handle larger loads and volumes of data.
- Horizontal scaling is more cost-effective than vertical scaling.

Scaling Databases

- Databases are critical components of most applications and require scaling to handle increased traffic and data volumes.
- Scaling databases can be done using either horizontal scaling or vertical scaling.

Vertical Scaling for Databases

- Vertical scaling for databases involves upgrading the hardware of the database server, such as adding more CPU, RAM, or storage.
- Vertical scaling can be expensive and may not be feasible in the long term.

Horizontal Scaling for Databases

- Horizontal scaling for databases involves adding more database servers to a system and distributing the load across them.
- Horizontal scaling is more cost-effective and can handle larger volumes of data and traffic.

Sharding

- Sharding is a technique used in horizontal scaling to distribute data across multiple servers.
- Data is divided into smaller chunks called shards, which are distributed across multiple servers.
- Sharding allows for better performance and scalability in large databases.

Consistency and Availability

- Consistency and availability are two key factors to consider when scaling databases.
- In horizontal scaling, ensuring consistency can be challenging due to the distribution of data across multiple servers.
- Ensuring high availability can also be challenging, as failures in one server can affect the entire system.

Conclusion

- Scaling databases is critical for handling increased traffic and data volumes in applications.
- Vertical scaling involves upgrading the hardware of the database server and is limited by the capacity of the server.
- Horizontal scaling involves adding more database servers to a system and distributing the load across them, making it more cost-effective and scalable.
- Sharding can be used to distribute data across multiple servers, but consistency and availability can be challenging in horizontal scaling.

Sharding in MySQL

What is Sharding?

- Sharding is a technique used to split a large database into smaller, more manageable pieces called shards. Each shard is hosted on a separate server, allowing for distributed storage and processing of data.
- Benefits of sharding in MySQL, including scalability, performance, and availability.

When is Sharding Required?

- Large and growing data volume: As a database grows in size, it becomes increasingly difficult to manage and query the data. Sharding allows for the data to be split into more manageable chunks.
- Heavy read/write loads: If a database experiences a high volume of read and write requests, sharding can help distribute the load across multiple servers.
- High-availability requirements: Sharding can provide redundancy and failover capabilities to ensure high availability in the event of a server failure.
- Geo-distribution of data: If data is stored in multiple locations, sharding can help ensure that each location has its own shard for faster access.

How Does Sharding Work?

- Partitioning the data: The data is split into smaller pieces called shards based on a sharding key.
- Sharding key: The sharding key is a field or set of fields used to determine which shard a piece of data belongs to.
- Distribution of data across multiple servers: The shards are distributed across multiple servers to enable distributed storage and processing of data.

Sharding in MySQL

- Horizontal partitioning: The data is partitioned based on rows, with each shard containing a subset of rows from the original table.
- Vertical partitioning: The data is partitioned based on columns, with each shard containing a subset of columns from the original table.
- Hash-based sharding: The data is partitioned based on a hash function applied to the sharding key.

Horizontal Partitioning

- Distribution of data across multiple nodes: The data is split into equal-sized partitions, with each partition stored on a separate node.
- Advantages and disadvantages of horizontal partitioning, including scalability, ease of implementation, and increased complexity in queries.
- Sample implementation in MySQL: Using the PARTITION BY RANGE or PARTITION BY HASH clauses in a CREATE TABLE statement.

Vertical Partitioning

- Splitting the data based on columns: Each shard contains a subset of columns from the original table.
- Advantages and disadvantages of vertical partitioning, including reduced storage requirements and increased complexity in queries.
- Sample implementation in MySQL: Using the CREATE TABLE statement to create separate tables for each subset of columns.

Hash-Based Sharding

- Use of hashing algorithm to determine the shard: A hash function is applied to the sharding key to determine which shard the data belongs to.
- Advantages and disadvantages of hash-based sharding, including load balancing and increased complexity in queries.
- Sample implementation in MySQL: Using the PARTITION BY clause with the HASH method in a CREATE TABLE statement.

An Introduction to NoSQL Databases

What is a NoSQL database?

- NoSQL stands for "Not only SQL."
- NoSQL databases are designed to handle large volumes of unstructured and semi-structured data that do not fit well into traditional relational databases.
- NoSQL databases are highly scalable and can handle a variety of data types, including text, images, videos, and other multimedia content.

Characteristics of NoSQL databases

- No fixed schema: Unlike relational databases, NoSQL databases do not require a fixed schema, allowing for more flexible data modeling.
- Scalability: NoSQL databases are highly scalable and can handle large volumes of data and traffic.
- High availability: NoSQL databases are designed for high availability and can handle hardware and network failures with ease.
- Distributed architecture: NoSQL databases are designed to run on distributed systems, allowing for better performance and fault tolerance.

Types of NoSQL databases

- Document databases: Store data in documents, such as JSON or XML, and provide flexible data modeling.
- Key-value stores: Store data in a key-value format, where the key is used to retrieve the data.
- Column-family stores: Store data in columns rather than rows, allowing for better performance and scalability.
- Graph databases: Store data in nodes and edges, making them ideal for storing and querying complex relationships.

Use cases for NoSQL databases

- Web applications: NoSQL databases are highly scalable and can handle large volumes of traffic and user data, making them ideal for web applications.
- Big data: NoSQL databases are designed to handle large volumes of unstructured and semi-structured data, making them ideal for big data applications.
- Real-time analytics: NoSQL databases can handle real-time data streams and provide real-time analytics.
- IoT: NoSQL databases are ideal for IoT applications that generate large volumes of data and require real-time processing.

Advantages of NoSQL databases

- Highly scalable: NoSQL databases are designed to handle large volumes of data and traffic, making them highly scalable.
- Flexible data modeling: NoSQL databases do not require a fixed schema, allowing for more flexible data modeling.
- High availability: NoSQL databases are designed for high availability and can handle hardware and network failures with ease.
- Distributed architecture: NoSQL databases are designed to run on distributed systems, allowing for better performance and fault tolerance.

Disadvantages of NoSQL databases

- No support for SQL: NoSQL databases do not support SQL, which can make it challenging for developers who are used to working with SQL-based databases.
- Limited transaction support: NoSQL databases do not support ACID transactions, which can be a disadvantage for some applications.
- Limited query support: NoSQL databases have limited query support compared to SQL-based databases, which can make it challenging to perform complex queries.

Examples of NoSQL databases

- MongoDB: A popular document database that is highly scalable and provides flexible data modeling.
- Apache Cassandra: A highly scalable and fault-tolerant distributed database.
- Redis: A fast and flexible key-value store.
- Neo4j: A graph database that is ideal for storing and querying complex relationships.

Conclusion

- NoSQL databases are designed to handle large volumes of unstructured and semi-structured data that do not fit well into traditional relational databases.
- NoSQL databases are highly scalable, flexible, and designed for high availability and fault tolerance.
- The best solution will depend on the required use case

SQL	NoSQL
RELATIONAL DATABASE MANAGEMENT SYSTEM (RDBMS)	Non-relational or distributed database system.
These databases have fixed or static or predefined schema	They have dynamic schema
These databases are not suited for hierarchical data storage.	These databases are best suited for hierarchical data storage.
These databases are best suited for complex queries	These databases are not so good for complex queries
Vertically Scalable	Horizontally scalable
Follows ACID property	Follows CAP(consistency, availability, partition tolerance)
Examples: MySQL, PostgreSQL, Oracle, MS-SQL Server, etc	**Examples:** MongoDB, GraphQL, HBase, Neo4j, Cassandra, etc

Understanding the CAP Theorem

What is the CAP theorem?

- The CAP theorem is a concept that describes the trade-offs that exist in distributed systems.
- It was first introduced by computer scientist Eric Brewer in 2000.

The three components of the CAP theorem

- Consistency: Every read receives the most recent write or an error
- Availability: Every request receives a response, without guarantee that it contains the most recent version of the information.
- Partition tolerance: The system continues to function even when network partitions occur.

Understanding Consistency

- In a consistent system, every read operation returns the most recent write operation's value.
- This means that the system must wait until all nodes agree on the value before returning a response.
- Achieving consistency can be challenging in distributed systems because of network latency and other issues.

Understanding Availability

- In an available system, every request receives a response.
- The response may not contain the most recent version of the information, but the system still responds to the request.
- Achieving availability is important in systems that require real-time or near-real-time responses.

Understanding Partition Tolerance

- In a partition-tolerant system, the system continues to function even when network partitions occur.
- Network partitions occur when communication between nodes is lost, and the nodes are unable to communicate with each other.
- Achieving partition tolerance requires designing the system in such a way that it can handle network partitions.

The CAP theorem trade-offs

- According to the CAP theorem, a distributed system can only achieve two of the three components.
- Consistency and availability can be achieved at the expense of partition tolerance.
- Availability and partition tolerance can be achieved at the expense of consistency.
- Consistency and partition tolerance can be achieved at the expense of availability.

Examples of CAP theorem trade-offs

- Google's Bigtable: Achieves partition tolerance and availability but sacrifices consistency.
- Amazon's DynamoDB: Achieves partition tolerance and availability but sacrifices consistency in some cases.
- MongoDB: Achieves consistency and partition tolerance but may sacrifice availability.

Connect with me on LinkedIn

Please subscribe to our YouTube channel

Check out my GitHub profile

Follow me on Twitter(X)

Thank you